(2)

Causal Distributed Breakpoints

Jerry Fowler
Willy Zwaenepoel

Rice COMP TR90-107

February 1990

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251-1892

(713) 527-8101

This work was supported in part by the National Science Foundation under Grants CDA-8619893 and CCR-8716914, and by the Office of Naval Research under Contract N00014-88-K-0140.

Causal Distributed Breakpoints

Jerry Fowler
Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, Texas 77251-1892
P.O. Box 1892

To appear,
10th International Conference on Distributed Computing Systems
Paris, France

Abstract

A *causal distributed breakpoint* is initiated by a sequential breakpoint in one process of a distributed computation, and restores each process in the computation to its earliest state that reflects all events that "happened before" the breakpoint. A causal distributed breakpoint is the natural extension for distributed programs of the conventional notion of a breakpoint in a sequential program. We present an algorithm for finding the causal distributed breakpoint given a sequential breakpoint in one of the processes. *Approximately consistent checkpoint sets* are used for efficiently restoring each process to its state in a causal distributed breakpoint.

Causal distributed breakpoints assume deterministic processes that communicate solely by messages. The dependencies that arise from communication between processes are logged. Dependency logging and approximately consistent checkpoint sets have been implemented on a network of SUN workstations running the V-System. Overhead on the message passing primitives varies between between 1 and 14 percent for dependency logging. Execution time overhead for a 200×200 Gaussian elimination is less than 4 percent, and generates a dependency log of 288 kilobytes.

1 Introduction

At a breakpoint the programmer would like to see the program in a state that reflects all events that have had a causal effect on the state of the program at the breakpoint. With a sequential program, this is straightforward, since only events earlier in time have a causal effect on events later in time, and sequential execution totally orders all events. In distributed programs, however,

This work was supported in part by the National Science Foundation under Grants CDA-8619893 and CCR-8716914, and by the Office of Naval Research under Contract N00014-88-K-0140.

breakpoints are more complicated, since events are only partially ordered, and since it is impossible to obtain an instantaneous snapshot of the entire program.

A *causal distributed breakpoint* is the natural extension for distributed programs of the conventional notion of a breakpoint in a sequential program. A causal distributed breakpoint restores each process to the earliest state that reflects all events that *happened before* the breakpoint, according to Lamport's partial order of events in a distributed system [8]. In contrast, previous definitions of distributed breakpoints simply find *any* consistent global system state including the breakpoint.

We present an algorithm for finding the causal distributed breakpoint given a sequential breakpoint in some process. The implementation of a causal distributed breakpoint requires a facility for restarting and replaying the execution of individual processes by means of *dependency logging*: Each message carries with it an *event index* for the sending process at the time the message was sent. This facility for restart and replay requires that process execution be deterministic.

The logging of dependency information is essential, but the message *data* may or may not be logged. Logging the data allows quick restoration of processes to a particular state, but requires much larger logs. The use of *approximately consistent checkpoint sets* expedites restoration of process events, permitting the use of dependency logging for all messages except those sent during the time the checkpoint is being taken. This keeps the logs small.

We describe our model of distributed computation in Section 2 of this paper. We define causal distributed breakpoints and motivate their definition in Section 3. Section 4 contains the causal distributed breakpoint algorithm, and Section 5 shows how to restore the system to a causal distributed breakpoint using approximately consistent checkpoint sets. Some implementation experience is described in Section 6. In Section 7 we survey related work, and in Section 8 we present conclusions.

2 Distributed System Model

We consider a collection of deterministic processes communicating solely via messages. The execution of the individual processes is deterministic in the following sense: If two processes start in the same state and receive the same sequence of input messages, they terminate in the same state and send the same sequence of output messages. An *event* is the sending or the receipt of a message. An event is uniquely identified within the process in which it occurs by an *event index*, which is incremented by one at each event.

Events are partially ordered by Lamport's "happened before" relation, denoted $\rightarrow$, and defined [8]:

1. If a and b are events in the same process, and a occurs before b, then $a \rightarrow b$.
2. If a is the sending of a message and b is the receipt of the same message, then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Two events a and b are *concurrent* if and only if $a \not\rightarrow b$ and $b \not\rightarrow a$.

A *process state* is an observation of the state of a process at some point in its execution. The process states of a particular process are totally ordered with respect to each other and with respect to all events happening in that process. A process state of a particular process *reflects* an event occurring in that process if the event happens before that process state. A *system state* is a set of

process states, one for each process. A system state is *consistent* if and only if for each process i, if the state of process i in the system state reflects event σ_i, then the state of all other processes must reflect all events that happened before event σ_i. That is, all messages received by all processes must have been sent. Only consistent system states can occur during normal execution.

3 Causal Distributed Breakpoints

3.1 Definition

A causal distributed breakpoint is initiated by the occurrence of a breakpoint in the *breakpoint process*. The *breakpoint event* is defined as the latest event in the *breakpoint process* that happened before the breakpoint. A *causal distributed breakpoint* is defined as a system state that consists of,

1. for the breakpoint process, its state at the time of the breakpoint, *and*
2. for all other processes, the earliest process state that reflects all events in that process that happened before the breakpoint event.

Any causal distributed breakpoint is a consistent system state.

With a sequential breakpoint, the state of a sequential program reflects all events that occurred in physical time before the breakpoint. The naive extension of this notion to a distributed program—i.e., the state of all processes such that they reflect all events that occurred in physical time before the breakpoint—is not achievable, because it is not possible to take an instantaneous snapshot of the entire system. Nor is it appropriate, because events that precede the breakpoint in physical time can obscure the causal relationship between events in a distributed program. Causal distributed breakpoints extend the notion of a sequential breakpoint so that the system state reflects all events that happened before the breakpoint according to Lamport's "happened before" partial order, which is an observable ordering. In Figure 1, the execution of three processes is shown, with time going from left to right. The messages m_1 through m_6 have been exchanged between the processes. A breakpoint has occurred in Process 1 at the location marked "×." The causal distributed breakpoint for this breakpoint event consists of the process states at the intersection of line **A** with the lines representing the execution of each process.

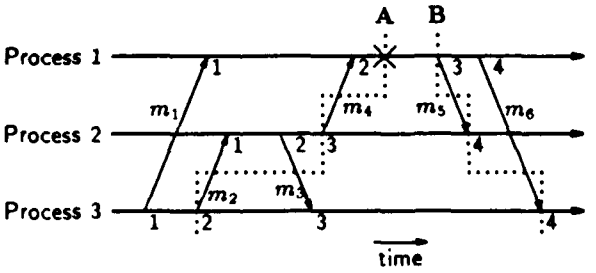

Figure 1 Distributed Breakpoints for Event 3 of Process 1.

3.2 Relation to Other Definitions

Miller and Choi [10], and Haban and Weigel [5] define a distributed breakpoint as *any* consistent system state that includes the breakpoint state. In an arbitrary consistent state, the processes other than the breakpoint process can be in any state that

1. is equal to or later than the earliest state that reflects all events that happened before the breakpoint event, and

2. is earlier than the first event that the breakpoint event happened before.

By contrast, in a causal distributed breakpoint, all processes are at the earliest state that reflects all events that happened before the breakpoint state. In Figure 1, a consistent system state is any system state consisting of process states between lines **A** and **B**. The causal distributed breakpoint is the system state indicated by line **A**.

The continued execution of a process beyond its state in the causal distributed breakpoint may obscure the causal relationships on which the state of the breakpoint process depends. For instance, in Figure 1, in the consistent system state denoted by line **B**, Process 3 has received message m_3. As a result, the state of Process 3 may have changed so as to destroy any evidence of why Process 3 sent message m_2, which had a causal effect on Process 1 at the breakpoint state.

4 The Causal Distributed Breakpoint Algorithm

Each message carries with it the event index of its sender at the time the message was sent. Dependency information is logged for each message received, and consists of the identification of the sending and receiving processes and the event indices corresponding to the sending and the receiving of the message.

The algorithm relies on the observation that if event τ_j in process j happened before event σ_i in process i, then all events from 0 to $\tau_j - 1$ in process j also happened before event σ_i in process i. Hence, it suffices to know for each event σ_i the latest event of each other process that happened before σ_i. We define the *dependency vector* [6] of event σ_i of process i, $\mathbf{DV}_i^\sigma$, as a vector

$$\mathbf{DV}_i^\sigma = \langle \delta_1, \delta_2, \delta_3, \ldots, \delta_n \rangle ,$$

where n is the total number of processes in the system. Component δ_i of the dependency vector of event σ_i of process i is always set to σ_i. Component δ_j of the dependency vector of event σ_i of process i is the largest event index that was received by process i in a message sent by process j. If process i has not received a message from process j, then δ_j is set to $\perp$, which is less than all possible event indices. The dependency vector of any event can be computed easily using the log for that process, given that for each message received the sender's identification and the send's event index are recorded in the log.

This dependency vector records the *direct* dependencies between events, resulting from the fact that send events happen before the corresponding receive events. However, *transitive* dependencies may also exist between events in two processes i and j, as a result, for instance, of a message sent from process i to process k and a message sent later from process k to process j. Such transitive dependencies must be taken into account when computing the causal distributed breakpoint.

Figure 2 shows the algorithm for finding the causal distributed breakpoint for event σ of process i. The algorithm computes for each process the latest event that happened before the breakpoint event. Procedure **CausalBkpt** starts by initializing the causal distributed breakpoint, **CDB**, to contain the breakpoint event σ for process i, and $\perp$ for all other processes. The algorithm then examines the dependency vector of each event it includes in **CDB** by recursively invoking **VisitEvent**. When **VisitEvent** finds an event index that is larger than the index of the corresponding process already included in **CDB**, **CDB** is altered to include the new, larger event index, and the dependency vector of that event is also examined. Thus, **VisitEvent** recursively includes all events that happened before the breakpoint event. Hence, when the recursion terminates, **CDB** holds the event indices of the causal distributed breakpoint.

Suppose that in Figure 1 we wish to find the causal distributed breakpoint for the state marked with an "×" in Process 1. **CDB** is initialized to $\langle 2, \perp, \perp \rangle$, and **VisitEvent** is invoked for event 2 of Process 1. The dependency vector $\mathbf{DV}_1^2$ for that event is $\langle 2, 3, 1 \rangle$. The event index for Process 2 in this dependency vector is 3, and is not included yet in **CDB**. Thus, **CDB** is set to $\langle 2, 3, \perp \rangle$ and **VisitEvent** is invoked with event 3 of Process 2. The dependency vector $\mathbf{DV}_2^3$ for that event is $\langle \perp, 3, 2 \rangle$. The event index for Process 3 in this dependency vector is 2 and is not included yet in **CDB**. Thus, **CDB** is set to $\langle 2, 3, 2 \rangle$ and **VisitEvent** is invoked with event 2 of Process 3. The dependency vector $\mathbf{DV}_3^2$ for that event is $\langle \perp, \perp, 2 \rangle$. All of its events have been included in **CDB**; hence this invocation of **VisitEvent** returns without making changes to **CDB**, as do both its

```
CausalBkpt (i : process, σ : event index)
   /* Causal distributed breakpoint for σ_i. */
   /* CDB holds the result. */
   for all k ≠ i
      CDB[k] = ⊥
   end for.
   CDB[i] = σ
   VisitEvent( i, σ )
end CausalBkpt.

VisitEvent (j : process, τ : event index)
   /* Put the dependencies of τ_j into CDB */
   for all k ≠ j
      α = DV_j^τ[k]
      if α > CDB[k]
         CDB[k] = α
         VisitEvent( k, α )
      endif
   end for.
end VisitEvent.
```

Figure 2 Causal Distributed Breakpoint Algorithm.

ancestor invocations. The final value of **CDB** is $\langle 2, 3, 2 \rangle$, which is the causal distributed breakpoint for the state marked with an "×" in Process 1.

A dual of the causal distributed breakpoint algorithm finds the state indicated by line **B** in Figure 1, which we refer to as the upper bound of execution with respect to the breakpoint event. The causal distributed breakpoint algorithm can be distributed by making **VisitEvent** a remote procedure call to the process named in the arguments and sending the current value of **CDB** with the remote procedure call.

In the worst case, the complexity of the algorithm is linear in the number of messages exchanged by the computation, since each dependency may invoke transitive dependencies on each process for which **VisitEvent** has not yet been invoked. **VisitEvent** is only invoked by a direct dependency, so it will never revisit the same event. A lower bound for complexity is $O(n)$, for a system of n processes, since each process must be examined at least once.

5 Approximately Consistent Checkpoint Sets

5.1 Motivation

Recreating a causal distributed breakpoint requires that each process i be restored to the process state immediately after the occurrence of the event with event index **CDB**[i].

If only the dependency information is recorded in the log, then every process must be restarted from its initial state. Since the order of receipt of messages is recorded in the log, all messages can be replayed to each process in the same order as in the original execution. Restoration is complete when each process i's event index equals **CDB**[i].

If the message data as well as the dependency information are recorded in the log for each message received, and processes take occasional checkpoints, then it suffices to restart each process i to the state recorded in its latest checkpoint before **CDB**[i], and replay from the log the messages received since that checkpoint until the event index equals **CDB**[i].

Recording the message data for each message received may result in impractically large logs. Recording only the dependency information produces much smaller logs, but leads to potentially long restoration times. *Approximately consistent checkpoint sets* are used to reduce the size of the logs while keeping restoration times short.

5.2 Method

A *checkpoint set* comprises a checkpoint for each process in the distributed computation. To establish an approximately consistent checkpoint set, a two-phase protocol is used. The initiator of a checkpoint broadcasts an out-of-band *checkpoint request* to all processes in the computation. The checkpoint request is retransmitted until an acknowledgment is received from all processes. A monotonically increasing checkpoint identifier is transmitted in the checkpoint request message to act as a sequence number for duplicate suppression among checkpoint requests.

Upon receiving a checkpoint request, a process performs a checkpoint and sends the event index recorded in its checkpoint to the coordinator in its acknowledgment of the checkpoint request. When the initiator has received acknowledgments from all processes in the computation, it performs a checkpoint and broadcasts to all processes an out-of-band *checkpoint confirmation* message that gives the event index recorded for each process in the checkpoint set.

In the interval between its checkpoint and its receipt of the checkpoint confirmation message, each process that receives a message records not only the dependencies, but also the data contained in the message. After the confirmation message is received, each process compares the send event index of each received message with the sender's checkpoint event index received in the checkpoint confirmation message. If the event index in the message is smaller, the data in the message are recorded along with the dependency information. Otherwise, only the dependency information is recorded.

To recreate a particular process state, it is never necessary to restart any process from a checkpoint earlier than its checkpoint in the most recent approximately consistent checkpoint set that precedes the state to be restored. The data of any message sent after a checkpoint in the set can be recreated by restarting from the checkpoint. Because all message dependencies are logged, the message can be replayed to the receiver in the order that was recorded during the original execution. Furthermore, although the data of any message sent before the checkpoint in the set cannot be recreated by restarting from the checkpoint, its data are available in the receiver's log.

The checkpoints in an approximately consistent checkpoint set are not necessarily consistent, because a process i can complete its checkpoint and then send a message to process j, which may be received before process j completes its checkpoint. The receipt of the message is recorded in j's checkpoint, but its sending is not recorded in i's checkpoint. Although the set of checkpoints is inconsistent, process i can deterministically execute forward from its checkpoint to the sending of the message. Hence, the term *approximately consistent* checkpoint set: The states recorded in the checkpoint set may not be a consistent system state, but they are a system state from which a consistent state can be recovered.

If the time between the completion of a checkpoint and the receipt of the checkpoint confirmation message is short compared to the interval between checkpoints, relatively few messages need to have their data logged, resulting in much smaller logs. However, restoration times should also be short, since it suffices to restart each process from its checkpoint in the approximately consistent checkpoint set preceding the state to be restored. To further reduce the amount of storage required to support causal distributed breakpoint, any checkpoint set except the initial state can safely be deleted, without impairing the ability to restore any event, albeit at the expense of slower restoration times for some system states.

6 Implementation Experience

6.1 Implementation

Causal distributed breakpoint has been implemented under the V-System [2]. Each participating host runs a modified V-System kernel, a *logging server* process managing the logging of messages received by the host, and a *checkpoint server* process managing checkpoint recording for the host.

As messages are received by a process, they are saved in the *message buffer* by the kernel. This buffer is stored in the volatile memory of the local logging server process, which periodically writes the contents of this buffer to the message log file on the shared network storage server. There is a separate message log file for each host.

The checkpoint server implements *full* and *incremental* process checkpoints. A full checkpoint writes the entire address space to the checkpoint file, and is used for the first checkpoint of each

Table 1

Performance of common V-System communication primitives with and without logging

Operation	Elapsed Time (milliseconds)		
	With Data	Dependencies Only	Without Logging
`Send-Receive-Reply`	1.63	1.57	1.37
`Send`(1K)-`Receive-Reply`	3.31	2.94	2.73
`MoveTo`(64K)	89.0	88.7	87.8

process. Thereafter, an incremental checkpoint is used to write only the pages of the address that have been modified since the last checkpoint. *Precopying* is used to limit the effect of checkpointing on execution [12]. In precopying, the process is allowed to continue executing while initial passes over the address space are made to write the necessary pages to disk. The process is then "frozen," and the pages that were changed since having been written in the last precopying pass are rewritten to disk. As a result, most of the checkpointing activity occurs concurrently with process execution.

The resulting implementation is efficient, yet kernel modifications are minor. The most performance-critical operation, recording dependencies in the volatile log, is the only one performed entirely in the kernel. In addition to several small changes to the internal operation of some existing kernel primitives, only four new primitives to support checkpointing and five new primitives to support logging were added to the kernel.

6.2 Performance

We measured performance on a group of diskless SUN-3/60 workstation, which have 20-megahertz Motorola MC68020 processors. They are connected by a 10 megabit per second Ethernet to a V-System file server running on a SUN-3/160, with a 16-megahertz MC68020 processor and a Fujitsu Eagle disk.

The overhead of message logging, with and without logging message data, for common V-System communication operations is given in Table 1. The performance is shown for a `Send-Receive-Reply` of a 32-byte message, `Send-Receive-Reply` of a 32-byte message with a 1-kilobyte appended data segment, and for a `MoveTo` bulk data transfer operation of 64 kilobytes. All times were measured at the user-level, and show the elapsed time between invoking the operation and its completion. The difference between no logging and logging the dependencies for `Send-Receive-Reply` (with and without appended data) is approximately 200 microseconds, indicating a fixed cost of approximately 100 microseconds per logging operation. The additional overhead for logging the message data stems from copying the message (and its appended segment, if any) in the log. The fact that the differences between the three numbers for `MoveTo` are so small is because these bulk data transfers are implemented as blasts of packets without intervening acknowledgments [13]. As a result, logging occurs in parallel with the transmission and reception of the next packet, so logging has almost no effect on the elapsed time of the operation.

The overhead of checkpointing in terms of elapsed time is on the order of 3 seconds per megabyte of address space. Since most of this overhead occurs in parallel with program execution, its effect on program execution time is small.

Table 2

Performance of 200 × 200 Gaussian elimination

	Without Logging	With Data	Dependencies Only	Approximate Checkpoints
Time (seconds)	86.7	87.7	87.3	89.7
Log (kilobytes)	-	2225	238	288

We ran a program performing Gaussian elimination with partial pivoting on a given $n \times n$ matrix of floating point numbers, with no logging, with logging only the dependency information, and with logging both the dependency information and the message data. This program employs a relatively large amount of communication, $O(n^2)$ for an $n \times n$ matrix.

The results of this experiment are shown in Table 2. With six slave processors computing on a 200 × 200 matrix, the execution time increased from 86.7 seconds without any logging, to 87.3 with logging dependencies (a 0.7 percent increase), and 87.7 with logging both message data and dependency information (a total 1.2 percent increase). The total size of the logs for all processes was approximately 240 kilobytes without message data, and 2.2 megabytes with message data. With approximately consistent checkpointing every 30 seconds, the overall execution time is increased by roughly 4 percent. The effect of approximately consistent checkpointing on the size of dependency logs is run-dependent, since the checkpoints are initiated asynchronously with respect to the computation, and both message frequency and message size during the checkpoint interval are factors in the size of the logs. In the Gauss experiments, the increase in log size due to data logging during checkpoint intervals is 50 kilobytes on the average.

7 Related Work

Miller and Choi [10] adapt Chandy and Lamport's distributed snapshots [1] to distributed breakpointing. Haban and Weigel [5] use a similar approach to that of Miller and Choi for generating interactive breakpoints. Cooper [4] broadcasts a halt request out-of-band, which produces a consistent system state when used during normal execution, or in replay from logs containing dependencies only. However, none of these methods gives the earliest system state that reflects all events that happened before the breakpoint event, as does causal distributed breakpoint. In fairness to the work of Miller and Choi, and that of Haban and Weigel, an implementation using their definitions of distributed breakpoint does not require that process execution be deterministic, as ours does.

In a later paper, Choi et al. develop the notion of a *before* and an *after* state in a parallel program, relative to a node in a parallel dynamic program dependency graph [3]. Their *before* state resembles our notion of causal distributed breakpoint. However, they do not address the issue of how to construct the dynamic dependency graph, which we do by including the event index of the sender in each message. Furthermore, their algorithm computes the *before* state for all nodes in the graph, while ours computes the causal distributed breakpoint for a single process state. Their paper reports no implementation results.

LeBlanc and Mellor-Crummey observe in Instant Replay that the size of the logs necessary to replay may be a concern [9]. They point out that it is not necessary to store the message data if all message communication can be described as serializable reads and writes of shared objects. Our

method of logging only dependency information is similar to theirs, although not restricted to any particular form of read/write access. Our use of approximately consistent checkpoint sets improves replay response time.

Causal dependency tracking by including the event index of the sender in each message, and the notion of a dependency vector are due to the work of Johnson and Zwaenepoel on optimistic message logging for fault tolerance [6]. The goal of replay in optimistic fault tolerance is to achieve a maximum recoverable system state. As a result, they use process state interval indices, incremented only upon receipt of a message, instead of our event indices, which are incremented for each event. Their checkpoint and recovery method requires that they log both dependencies *and* message data. An alternative method of dependency tracking by including the full dependency vector with each message sent was introduced earlier by Strom and Yemini [11].

The approximately consistent checkpoint set bears some similarity to the Chandy-Lamport snapshot algorithm [1] and the consistent checkpoints of Koo and Toueg [7]. Chandy and Lamport assume reliable communication channels, where we do not. We assume deterministic process execution, which they do not. We use these checkpoints for replay, while they use the checkpoints and the channel states for detecting stable properties of computations. As a result, in the snapshot algorithm, messages sent by a process after its checkpoint must not be received before the checkpoint of the recipient. For replay of deterministic processes, this is allowable if the original order of message receipt is reproduced. Koo and Toueg generate consistent checkpoints, at the expense of prohibiting message traffic during the execution of the checkpoint algorithm. Our algorithm does not prohibit message traffic at any time, but requires logging of the dependency information and some message data, and assumes deterministic processes. Our algorithm does not produce consistent checkpoints, but achieves a consistent system state through re-execution.

8 Conclusion

We have defined causal distributed breakpoint, which naturally extends the notion of a sequential breakpoint to distributed systems, where only a partial ordering between events can be observed. When one process of a distributed computation is halted at a breakpoint, the causal distributed breakpoint shows the other processes in the computation in their earliest state that reflects all events that happened before the breakpoint. Previous distributed breakpoint definitions allow any consistent system state including the breakpoint. This may obscure the causal relationship between the states of other processes and the state of the breakpoint process.

We have also introduced the notion of approximately consistent checkpoint sets, which allow quick restoration of process states with relatively small logs. Our implementation shows that the overhead of dependency logging has little effect on program execution time, and that log sizes using approximately consistent checkpoint sets are acceptable.

Acknowledgments

We wish to thank Dave Johnson for his many useful criticisms of this work, as well as Rick Bubenik, John Carter, Mootaz Elnozahy, Pete Keleher, and Mark Mazina, for their comments on earlier drafts of this paper.

References

[1] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[2] David R. Cheriton and Willy Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140. ACM, October 1983.

[3] Jong-Deok Choi, Barton P. Miller, and Robert Netzer. Techniques for debugging parallel programs with flowback analysis. Technical Report TR 786, University of Wisconsin, Madison, WI, August 1988.

[4] Robert Cooper. Pilgrim: A debugger for distributed systems. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 458–465. IEEE, September 1987.

[5] Dieter Haban and Wolfgang Weigel. Global events and global breakpoints in distributed systems. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences, Vol II, Software Track*, pages 166–175. IEEE Computer Society, January 1988.

[6] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181. ACM, August 1988. May 1988.

[7] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[8] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[9] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[10] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 141–150, June 1988.

[11] Robert E. Strom and Shaula Yemini. Optimistic recovery: An asynchronous approach to fault-tolerance in distributed systems. In *The Fourteenth International Conference on Fault-Tolerant Computing: Digest of Papers*, pages 374–379. IEEE Computer Society, June 1984.

[12] Marvin N. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 2–12. ACM, December 1985.

[13] Willy Zwaenepoel. Protocols for large data transfers over local area networks. In *Proceedings of the 9th Data Communications Symposium*, pages 22–32. IEEE Computer Society, September 1985.